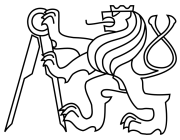
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Improvement of Contactless Smart Card Emulator in FPGA
Student:	Denis Titov
Supervisor:	Ing. Stanislav Jeřábek
Study Programme:	Informatics
Study Branch:	Information Technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2017/18

Instructions

Study the ISO 14443 standard on contactless Smart Cards.
Study the existing emulator, made as a result of master thesis [1].
Make emulated UID, SAKC answer, and ATQA answer configurable at run time without changing the VHDL source code.
Prepare the existing emulator for receiving proprietary protocol commands in accordance to the fourth part of the ISO 14443 standard.
The data bytes will be processed by a processor, either connected to the FPGA externally, or integrated in the FPGA (e.g., Xilinx Zynq).
Demonstrate this by a practical test of Card Select and by some custom commands in accordance to the standard.

References

[1] Jeřábek, Stanislav. Emulátor bezkontaktní čipové karty v FPGA. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague November 18, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Bachelor's thesis

Improvement of Contactless Smart Card Emulator in FPGA

Denis Titov

Supervisor: Ing. Stanislav Jeřábek

16th May 2017

Acknowledgements

To my family for their constant support and to my supervisor, Stanislav Jeřábek, for his priceless help and endless patience

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Titov, Denis. *Improvement of Contactless Smart Card Emulator in FPGA*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato práce popisuje vylepšení přidaná k již existujícímu emulátoru čipových karet v FPGA. Byla přidána možnost rekonfigurace emulátoru za běhu a také podpora pro proprietární protokoly podle normy ISO/IEC 14443-4. Pro splnění požadavků této normy byl zvolen hybridní přístup se dvěma komponentami - FPGA a mikrokontroler. Komponenty jsou propojeny pomocí SPI.

Klíčová slova FPGA,mikrokontrolér,ISO/IEC 14443-4,čipové karty,emulace

Abstract

This thesis describes improvements to existing FPGA smart-card emulator. Run-time configuration was implemented and foundation for ISO/IEC 14443-4 based proprietary protocols support was laid out. To comply with these protocols requirements, hybrid approach with two components – FPGA and MCU – was used. Components are connected by SPI.

Keywords FPGA,Microcontroller,ISO/IEC 14443-4,Smart card,Emulation

Contents

List of Figures

List of Tables

Introduction

The emulator [1], this work is based on was implemented entirely on Xilinx Spartan 6 FPGA. This decision have lead to significant performance advantage over "typical" microcontroller based implementations. The emulator could provide reply in 40 ns since complete command arrival, while standard requires this period to be less than 67 000 ns. At the same time, the emulator have used relatively small amount of resources (about 14% of Xilinx Nexys 3 board).

However, it could only emulate cards with very basic functionality, implementing only card selection protocol, defined in ISO/IEC 14443-3 [2]. While perfectly sufficient to emulate some types of smart cards (for example, Mifare Classic, which was used in demonstration of original emulator capabilities).

Moreover, it was not configurable, as all parameters were hard-coded as vhdl constants. Even though user could code up to 4 different configurations to select from, anything extra have required editing of emulator source and flashing the board with the updated firmware. Taken together with synthesis constant delays in work, caused by waiting for synthesis to finish, this procedure can become very cumbersome.

Goal of the work

In this work we aimed to, at least partially, rectify these two factors.

Implementation of ISO/IEC 14443-4 standard seems to be a good starting point for providing various proprietary protocols support. For example, MIFARE DESFire and MIFARE Plus cards' protocols are of such nature [3]. However, its resource (and, especially, memory) requirements are much higher, than ISO/IEC 14443-3 has. While low-level block of the third part don't size of 5 bytes (plus, 2-byte CRC, which doesn't need to be stored), the 14443-4 standard frames can be up to 256 bytes long.

While limiting set of available for emulation cards by frame size is an option, it is not the best solution. Moreover, this situation would lead to large increase in FPGA space usage, which also goes against this emulator principles.

As result, we have settled for the alternative solution: turning FPGA emulator into a hybrid system, where large high-level data blocks are processed by microcontroller unit (MCU). In this work we implement data forwarding between those units, preparing them for future implementation of proprietary protocols.

The configurability issue should be addressed by addition of external interface, over which user can change various emulator parameters in the runtime.

Chapter 1

Analysis

About previous work

Even though FPGA implementation had given emulator a significant speed advantage over microcontroller based implementations, it have also had lead to several drawbacks. This work targets two of them: limited spectrum of potentially emulated cards and lack of runtime reconfiguration.

Lack of support for higher-level operations (including ones, defined in the fourth part of the standard [4]) have significantly limited set of smart cards that could be emulated.

The emulator flexibility was also limited by lack of configurability. The user could choose from 4 hard-coded configurations by changing position of switches on used dev. board. Any change to these configurations is only possible after editing vhdl source code, resynthesizing it and reprogramming the FPGA. This process is rather inconvenient and can take a significant amount of time.

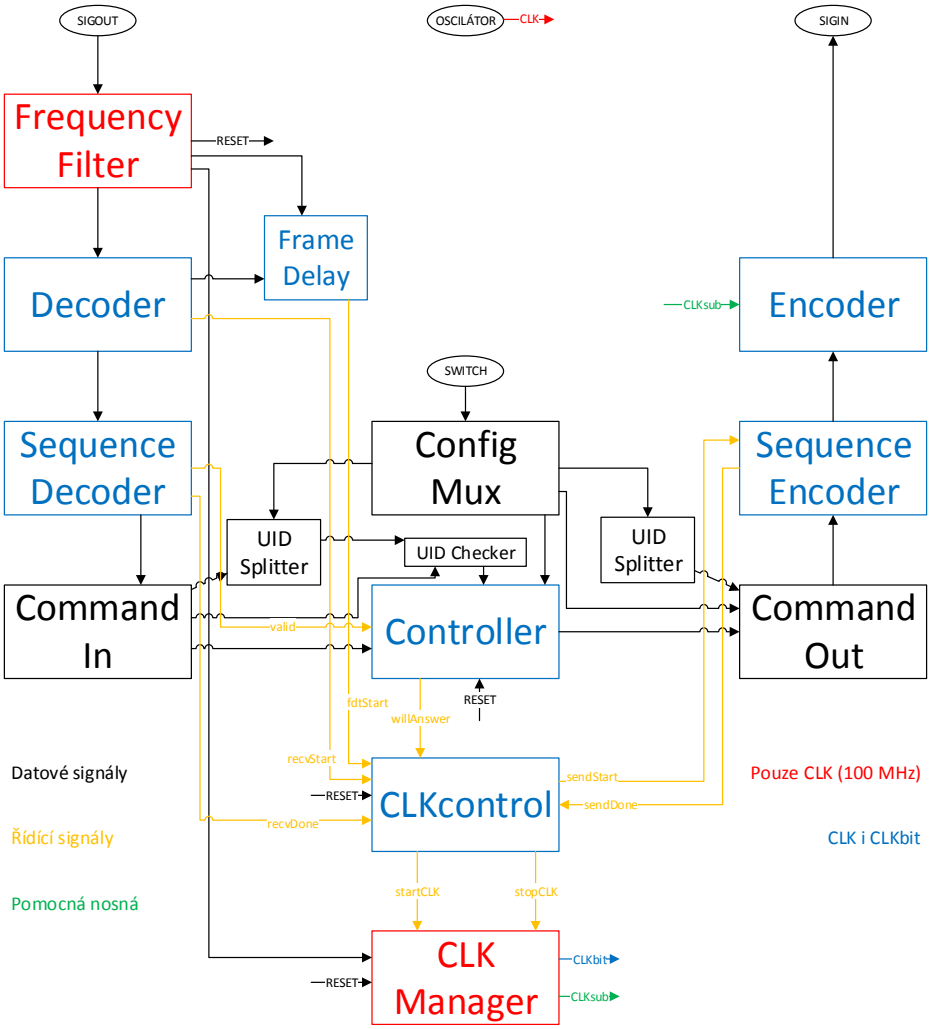


Figure 1.1: Block diagram of the old emulator

Introduction to ISO/IEC 14443-4

The fourth part of ISO/IEC 14443 standard defines half-duplex block transmission protocol, where card reader, further denoted as PCD, serves as an active part. This protocol follows OSI reference model, combining properties of data link and session levels for minimal overhead. It is used over physical level protocol defined in the third part [2]. The Fig. 1 shows how these protocols are connected.

Following PICC activation sequence from ISO/IEC 14443-3 [2], PCD can find out whether card, also denoted as PICC, supports more high-level operations, or not by analysing contents of SAK byte. Even though the standard leaves ground for any proprietary protocols, we would only consider cards, that are ISO/IEC 14443-4 [4] compliant.

If `PCD` determines that card implements ISO/IEC 14443-4, it can proceed with protocol initialization by sending `RATS` block. Apart from signalizing start of higher-level interaction, the `RATS` block also transmits two important parameters: `FSDI` and `CID`.

The `FSDI` is an encoded maximal length of block, that card reader can accept. You can find encoding details in Tab. 1

`CID` is only used in cases, when there are several active cards on one reader. It is a temporary "card address", valid as long as this card is activated. Each card in the field should have an unique `CID`. The `CID = 15` is a reserved combination (RFU).

PICC should reply with `ATS` block. The `ATS` consists of length byte (`TL`), optional format byte (`T0`), interface bytes (`TA(1)`, `TB(1)`, `TC(1)`) and also historical bytes (`T1` to `Tk`). The frame is finished by 2-byte CRC.

The `TL` contains length of all the whole block, except from CRC. Length of `ATS` shouldn't exceed `FSD`, so maximal value of `TL` is `FSD - 2`. If `TL = X"01"`, then all the option bytes are absent and `PCD` should assume default parameters.

FSDI or FSCI	Length in bytes (FSD or FSC)
0	16
1	24
2	32
3	40
4	48
5	64
6	96
7	128
8	256
9 - F	Reserved

Table 1.1: Coding of FSCI and FSDI

PCB	CID	NAD	Data 1	.	Data N	CRC1	CRC2

Table 1.2: General structure of ISO/IEC 14443-4 protocol block

This emulator doesn't support these options, but we still should mention one of them: `FSCI`. It is a complete analogue of `FSDI`, except for that it codes maximal block length acceptable *by card*. The default value of `FSCI` is 2, which corresponds to 32 bytes.

The CRC algorithm used is the same, as in data transmission protocol that we'll describe later. It is defined in the third part of standard, under the name of `CRC_A` [2].

An optional exchange of PPS request (from `PCD`) and PPS reply (from `PICC`) blocks can happen at that moment, changing several configuration parameters. We wouldn't describe these blocks in detail, as these parameters are not supported as a part of this work.

After that the protocol initialization is considered to be complete and both parts can exchange data blocks, structure of which is covered in the next subsection. The protocol operation scenarios are demonstrated in the end of this section.

`PCD` can terminate communication at any moment by sending `DESELECT` block. The card should reply with `DESELECT`. After `DESELECT` `PICC` should move into `HALT` state.

Block types and their structure

Each block consists of prologue field, informational field and epilogue field. Prologue and epilogue are mandatory, information is optional. This structure is demonstrated in Tab. 1.2

The prologue can have up to 3 bytes: Protocol Control Byte (mandatory), Card Identifier (optional) and Node Address (optional).

`PCB` codes block type, block number and presence/absence of other prologue bytes. There are 3 fundamental block types: I-blocks, used to transmit application data, R-blocks, used for positive and negative acknowledgements, and S-blocks, used for `DESELECT` and `WTX` (wait time extension) commands.

The next prologue field - `CID` - is used for `PICC` identification for environments with multiple active cards. In that case, each `PICC` replies either to blocks with their own `CID`, or to blocks, where `CID` is not present. Each `PICC`'s `CID` is set by the `PCD` during card activation.

`NAD` field is used for addressing multiple logical links under one physical connection. The structure of `NAD` byte and usage of such links are described in ISO 7816-3 standard [5].

I-blocks are the only blocks, which can have a `NAD` header. Several I-blocks can be "chained" for transmitting application level data, bigger than maximal

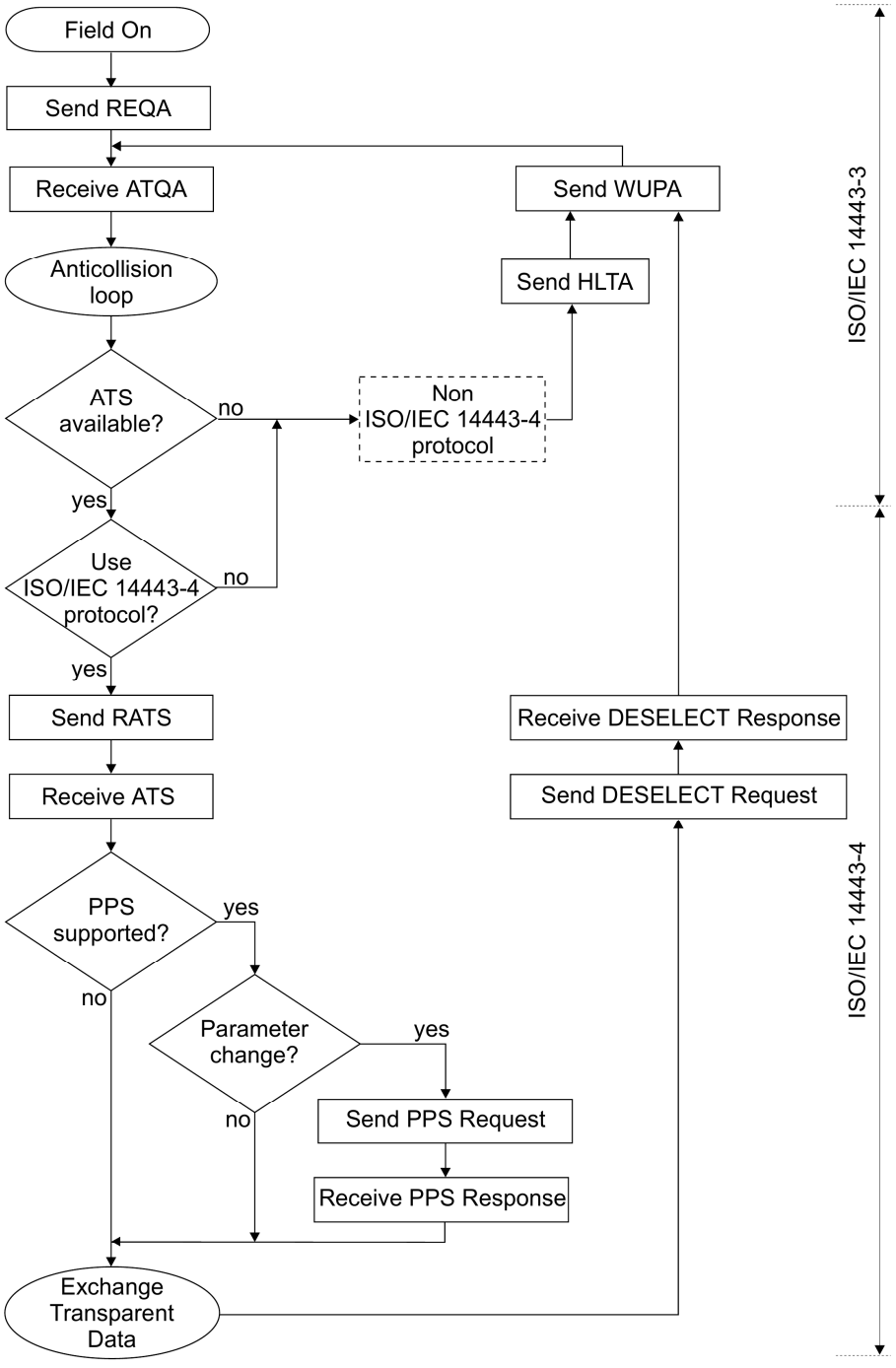


Figure 1.2: States of ISO/IEC 14443

allowed block size. The chaining is enabled by setting special bit of PCB to '1' for all chained blocks, apart from the last.

The standard defines two possible R-blocks: ACK and NAK. R-blocks cannot contain information field. ACK frames are used to acknowledge receiving part of a chained frame (except from the last part, which is acknowledged by proper answer: I-block). NAK frames are only used by PCD for notifying PICC about communication errors. The detailed rules are described in the fourth part of original standard [4].

Two S-block types are defined in standard: DESELECT and WTX. DESELECT blocks are used for communication termination and are initially issued by PCD. On the other hand, WTX (Wait Time Extension) blocks are initially sent by PICC. Their purpose is to tell PCD that answer calculation is not finished yet.

S-blocks should always be issued in pairs: request and identical reply. If one of these blocks doesn't arrive, error handling rules of the standard are employed [4].

Both WTX and DESELECT blocks should contain 1-byte information field (WTXM), which codes requested amount of time to wait. The standard describes this field coding in more detailed manner.

The epilogue consists of 2-byte control sum, algorithm of which is defined in ISO 14443-3 under the name of CRC_A [2]

Examples of protocol operation

PCD	Dir	PICC	Step Description
I(0,0)	→		PCD starts transmission, BNO=0, no chaining
	←	I(0,0)	PICC answers with data
I(1,0)	→		BNO changes after each block
	←	S(WTX) Request	PICC cannot reply in time, requesting WTX
S(WTX) Response	→		Request approved
	←	I(1,0)	
I(0,1)	→		Chained block
	←	R(ACK)(0)	PICC confirms received block
I(1,0)	→		Last block in chain
	←	I(0,0)	Each party has its own BNO
S(DESELECT)	→		PCD requests termination
	←	S(DESELECT)	PICC agrees, connection is closed

Table 1.3: Normal operation of ISO block transmission protocol

PCD	Dir	PICC	Step Description
I(0)			No data received
			As result, reply time-out
I(0)	$\not\rightarrow$		PCD retransmits data, but with errors
	$\leftarrow$	R(NAK)(0)	PICC asks for another try
I(0)	$\rightarrow$		The data have finally arrived

Table 1.4: Error handling in ISO-14443-4

Configuring the emulator

One of this thesis tasks is to improve emulator flexibility, by introducing runtime configuration. There are several parameters, which identify either type of emulated card (e.g. `SAKC` byte), or its instance (for example, `UID`). Previously they have been saved in a set of vhdl constants, making it necessary to edit source code and reprogram FPGA in case of any changes. These constants have been replaced with a register file, contents of which can be altered via dedicated RS232 interface. A very simple protocol is used for parameters encoding. Its frames' structure is shown at Fig. 1

The header consists of 2 parts: `ConfKey` and `ValueLen`. The former codes key of parameter to be retrieved or changed. The later part, `ValueLen`, serves two purposes: if it has value of `0x0`, then the current configuration of emulator is not changed and current value of parameter is sent back. In other case, emulator expects to receive `ValueLen` bytes of new value. After that it updates respective parameter and replies with the new value.

The reply frame always has `ConfKey` of requested or updated parameter and its actual value.

To simplify this protocol, no error detection/correction techniques have been employed. However, user can verify correctness of change by the emulator reply.

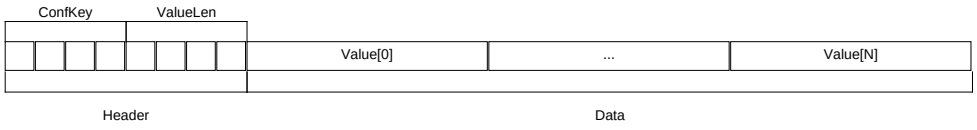


Figure 1.3: Structure of configuration frame

Parameter	Key	Coding
ATQA	0x1	2 bytes of binary data
FSCI	0x2	1 byte unsigned integer, encoding FSC value by ISO standard
FSDI	0x3	1 byte unsigned integer, encoding FSD value by ISO standard
SAKC	0x4	1 byte of binary data
UID	0x5	4, 7 or 10 bytes of binary data
MCUTimeout	0x6	1 byte unsigned integer

Table 1.5: List of parameters and their keys

MCUProto data forwarding protocol

In this section a protocol for communication between FPGA and MCU parts of the emulator is described. This protocol – `MCUProto` – is a byte-oriented protocol with half-duplex architecture, where active part is played by FPGA.

The protocol life cycle corresponds to the one of "Half-duplex block transmission protocol", defined in the fourth part of ISO standard [4]. For disambiguation purposes, ISO protocol messages are further denoted as "ISO blocks" or just "blocks", while `MCUProto` messages are denoted as "`MCUProto` frames", or just "frames".

Frame structure

Each frame consists of 2-byte header, up to 15 bytes of data and 2 bytes of CRC. Header contains `BNO` of corresponding ISO block, number of this frame (`FNO`), frame `OpCode` and length of transmitted data (header and CRC not included). Data area contains forwarded ISO data. For the simplicity purposes, CRC uses the same algorithm, as ISO protocol - `CRC_A`, defined in ISO 14443-3 [2].

`MCU_CONF`, `MCU_IDENY`, `ISO_ACK` and `ISO_NAK` frames don't have data area, their `DataLens` should be equal to `0x0`. `ISO_SELECT`, `ISO_DESELECT`, `ISO_WTX`, `MCU_CONF` and `MCU_IDENY` frames' `BNO` bits are not used, since their ISO counterparts either don't have `BNO` (as, for example, `DESELECT`), or they do not correspond to any ISO blocks at all (as `MCU_CONF`). For these frames, the `BNO` bit should always have `'0'` value.

Length and contents of frames data areas are described in Tab. 1.

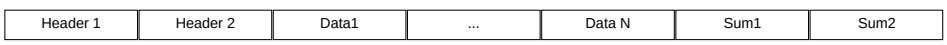


Figure 1.4: Structure of MCUProto frame

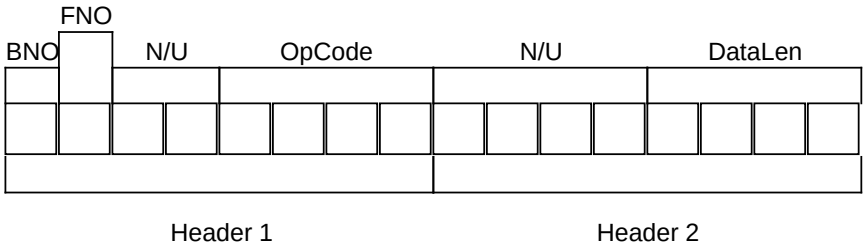


Figure 1.5: Detailed structure of MCUProto header

Name	Code	Description
ISO_SELECT	0x1	Start transmission
ISO_DESELECT	0x2	Stop transmission
ISO_WTX	0x3	Ask for reply time extension
ISO_ACK	0x4	Forward ISO acknowledge
ISO_NAK	0x5	Forward negative ISO acknowledge
ISO_I_CONT	0x6	Forward part of I-block
ISO_I_END	0x7	Forward last part of I-block
ISO_ICHAIN_CONT	0x8	Forward part of chained I-block
ISO_ICHAIN_END	0x9	Forward last part of chained I-block
MCU_CONF	0xA	Confirm received MCUProto frame
MCU_IDENY	0xB	Drop current I-block, as invalid

Table 1.6: List of MCUProto operations

Name	Len	Content
ISO_SELECT	1 byte	FSDI of card reader
ISO_DESELECT	1 byte	WTXM of respective DESELECT
ISO_WTX	1 byte	WTXM of respective WTX
ISO_I_CONT	Variable	Part of respective I-block data
ISO_I_END	Variable	Part of respective I-block data
ISO_ICHAIN_CONT	Variable	Part of respective I-block data
ISO_ICHAIN_END	Variable	Part of respective I-block data

Table 1.7: Format of data area in MCUProto frames

Protocol operation

The communication starts with FPGA issuing ISO_SELECT frame, having FNO = 0. It happens when RATS block is received from PCD. MCU answers to ISO_SELECT with MCU_CONF, with the same FNO. After that, MCU is ready to receive forwarded data.

FNO (frame number) is inverted after each confirmed frame exchange. Passive actor (MCU) should reply with received FNO. As result, ISO_CONT and ISO_CHAIN_CONT frames, sent from MCU are confirmed by MCU_CONFs with inverse FNO, while FPGA → MCU transmissions are confirmed by equal FNO.

Communication is terminated by ISO_DESELECT frame (which, in turn, is sent after receiving DESELECT from PCD). The MCU can optionally reply with ISO_DESELECT frame. Whether to reply, or not is decided by emulator user, basing on the model of emulated card.

ISO R-blocks (ACK and NAK) are forwarded to MCU as they are. Their handling is dependent on context and complains with ISO-14443-4 protocol [4].

ISO I-blocks are forwarded using cut-through switching technique. Only data part is forwarded – headers and control sums are handled on FPGA. All data chunks, except for the last are wrapped into ISO_I_CONT frames. ISO_I_CONT frames should be confirmed by MCU_CONF. The last chunk in block is wrapped into ISO_I_END instead. It doesn't require confirmation, instead expecting actual calculated answer.

The negative confirmation is coded by MCU_CONF frame with non-matching FNO (i.e. received FNO $\neq$ sent FNO for FPGA → MCU transmissions and received FNO = sent FNO for opposite direction). If FPGA doesn't receive confirmation until time-out (from 4 up to 64 protocol cycles, settable by user), the last frame should be resent. As MCU is a passive part, it doesn't keep track of time-outs. Should one occur, the FPGA will send MCU_CONF with non-matching FNO instead.

If received I-block is chained, then ISO_ICHAIN_CONT and ISO_ICHAIN_END are used instead of ISO_I_CONT and ISO_I_END respectively. Also, unlike their simple counterparts, ISO_ICHAIN_END frames should be confirmed by ISO_ACK frames with correct FNO ("correctness" is checked in the same way, as in case of MCU_CONF frames) and BNO (as ISO standard defines).

FPGA	MCU	Description
ISO_SELECT(0)		Connection initialization
	MCU_CONF(0)	Connection confirmation
ISO_I_END(1)		Forwarding short data block
	ISO_I_CONT(1)	Replying with longer block
MCU_CONF(0)		When going from FPGA to MCU, FNO checks are reversed
	ISO_I_END(0)	Finishing reply
ISO_ICHAIN_CONT(1)		Forwarding chained ISO block
	MCU_CONF(1)	
ISO_ICHAIN_END(0)		
	ISO_ACK(0)	End of chained frame is acknowledged by ISO standard
ISO_I_END(1)		End of blocks chain
	ISO_I_END(1)	And MCU replies
ISO_DESELECT(0)		PCD decided to finish communication
	ISO_DESELECT(0)	Confirming termination

Table 1.8: Example of MCUProto operation.
Notation: OPERATION_CODE(FNO)

Design considerations

The main restriction, which have influenced architecture and design the most, is lack of space for processed frame storage. According to standard, ISO protocol blocks can reach up to 256 bytes of size. This is clearly too much for register-based storage. At the same time, usage of RAM would make this emulator more platform dependent.

Even though limiting set of emulated cards by their maximal supported block sizes is an option, we have decided against it for two reasons: First, it decreases emulator flexibility, not only by limiting system capabilities, but also by increasing space requirements and, thus, limiting set of potentially supported FPGAs. And, second, supporting larger block sizes would increase space consumption rapidly, decreasing emulator users' abilities to implement application logic for proprietary protocol of their choice.

As result, we have decided to turn emulator into a hybrid system, consisting of both FPGA and microcontroller (MCU) units.

In this system, FPGA deals with "routine work" – encoding and decoding

of data, CRC verification and basic operation processing (e.g card selection). At the same time, all high-level frames are forwarded to MCU.

The space restrictions have naturally led us to use a cut-through switching technique, where data is forwarded before full retrieval. The detailed description of used forwarding method and reasoning behind taken decisions are presented in subsection "`MCUProto` frames" 1.

Some discussion on configuration protocol is presented in the "Emulator configuration" subsection 1. Finally, in the "MCU communication interface" 1 subsection reasoning behind interface choice is demonstrated.

MCUProto frames

Even though we implement our cut-through forwarding, we do not send data as soon, as it arrives, because doing so would increase either chance of partial data loss (if we will not use error detection), or utility overhead (if we would). However, at the same time we can't afford to store big blocks of data, as it would greatly increase surface area requirements of emulator. Moreover, storage of large data blocks would increase emulator response time because we have to wait until the whole block would arrive.

In attempt to balance these factors, we have chosen each frame to carry up to 15 bytes of data – size of ISO block with minimal `FSC` (taken without header, which is processed on FPGA). This data chunk is also accompanied by 2-byte header, which caries information about block type, `BNO` and length of attached data. 2-byte CRCs are not stored, but rather calculated on-demand.

To simplify protocol understanding most of commands just mirror ISO operations. However, some modifications had to be made. As a single I-block can contain data for several `MCUProto` frames, we needed to introduce our own frame chaining, which works not unlike ISO blocks level one. To handle this, I-blocks were mapped onto `ISO_I_CONT` and `ISO_I_END` frames, and `MCU_CONF` was introduced as analogue for `MCU_ACK` frame. Note that instead of creating analogue to `ISO_NAK` as well, we just use "incorrect" `MCU_CONF` frames.

`MCUProto` chaining works in an analogous way to its ISO analogue: a single data block is separated into a set of frames, where each of them, but last have `OpCode` of `ISO_I_CONT`, while the last has `ISO_I_END` instead. Each frame is confirmed by an `MCU_CONF`. Repetitions are tracked with the help of frame numbering by an `FNO` bit.

For processing of chained ISO blocks we have introduced matching pair of `OpCode`s instead: `ISO_I_CONT` and `ISO_I_END`, as it uses header space more effectively, than a distinguished chaining bit would.

The alternative solution – to reassemble chained frames on FPGA – have proven to be impossible to implement: Let's imagine a following setup: emulator replies to PCD with chain of I-blocks. If one block in the chain would arrive with error, then, according to the standard, PCD would ask us to retransmit it as it have required to store the whole chained frame. However,

start of the current block could have been rather long ago, so we wouldn't have this data available on FPGA any more. The MCU, on the other hand, wouldn't have any information about block borders. As result, it would become impossible to recover from such situation.

Another consequence of our cut-through processing led to appearance of `MCU_IDENY` frame. As we receive ISO block, byte by byte, we don't know if there have been any transmission errors. This can be checked only after CRC verification, which is possible only after receiving the whole block. At that time we would already have one or several `MCUProto` frames sent to MCU, so in case of CRC mismatch we need a way to indicate that data, we sent, was incorrect. The `MCU_IDENY` serves for this task.

Emulator configuration

This protocol was designed to be as minimal, and as simple as possible, aiming for use via simple client applications, or even general telnet client. The only step away from this principle is binary nature of the protocol, which is a necessary performance trade-off. For example, `ConfKey` is placed into the higher half-byte to make mental calculation of header easier – you just need to arithmetically add length to the character value of `ConfKey`.

For the same reasons, this protocol doesn't have any error detection techniques and the only way to validate configuration correctness is by comparison of expected value and reply.

MCU communication interface

Even though UART protocol seemed to be a natural solution for connecting emulator to MCU, it has proved to be too slow for this purpose. If we would use interface with baudrate of 115200 in 8-N-1 mode, the resulting data speed would be 92160 useful bits per second. At the same time, the standard [4] defines default waiting time between frames as approximately 4.8 ms and default frame size as 32 bytes. We need to transmit at least 64 bytes (incoming to MCU, and answer to emulator) but UART will be able to handle only 55 bytes in that period.

Even separation of answer into smaller portions to start transmission before receiving complete result would not help: the standard [2] defines length of 1 bit (`etu`) as `128*fc`, which is approximately equal to 9.3 microseconds. At the same time, transmission of single bit of data over UART would approximately take 10.85 microseconds, leaving us unable to "catch up".

This problem can be overcome by using faster communication interface. The SPI was chosen for its simplicity of implementation and operation. Due to having more strict time requirements and in connection to its role in `MCUProto`, FPGA emulator was chosen as an interface master. The working frequency

was selected to exceed `2*(128*fc)` (to be able to potentially process data "in real time") and equals to 250 kHz.

At this frequency, transmission of 1 bit would take 4 microseconds, enabling us to exchange up to 150 bytes during the standard `FWT`. It gives us more than enough time to transmit standard 32-byte data blocks, encoded in `MCUProto` frames, together with necessary utility frames.

Chapter 2

Implementation

General structure

As you could see in Fig. 1, the main data path of the old emulator version went through the following chain: `sequenceDecoder` – `commandIn` – `controller` – `sequenceEncoder`.

To implement high-level operations forwarding, we have implemented an alternative data path, which starts from `sequenceDecoder` and ends at `sequenceEncoder`. Which of these paths are used at the moment is determined by the `byteMode` signal, which is issued by the controller.

Paths are switched, when `controller` receives `RATS` command from the `commandIn` module. After that input commands are handled by `MCUDataIn` module instead. The `controller` does not participate in data handling. The only exception is `DESELECT` block handling, where controller should reset `byteMode`. Thus,`MCUDataIn` just sets a dummy input command (`NONE`) most of the time, making the only exception for `DESELECT`, by issuing command with the same name.

However, the main `MCUDataIn` task is preparation of arrived data for transmission to the MCU, while `MCUDataOut` converts `MCUProto` replies back to ISO blocks.

Error handling for `MCUProto` is implemented in the following modules: `MCUDataSend`, `FrameValidator` and `MCUDataRecv`. In particular, frame numbering is handled by the `FrameValidator` module, which provides this information to both `MCUDataSend` and `MCUDataReply`.

The emulator configuration is stored in `ConfStorage` module. Other `Conf*` modules are necessary for changing configuration over UART interface.

The Fig. 2 demonstrates this setup.

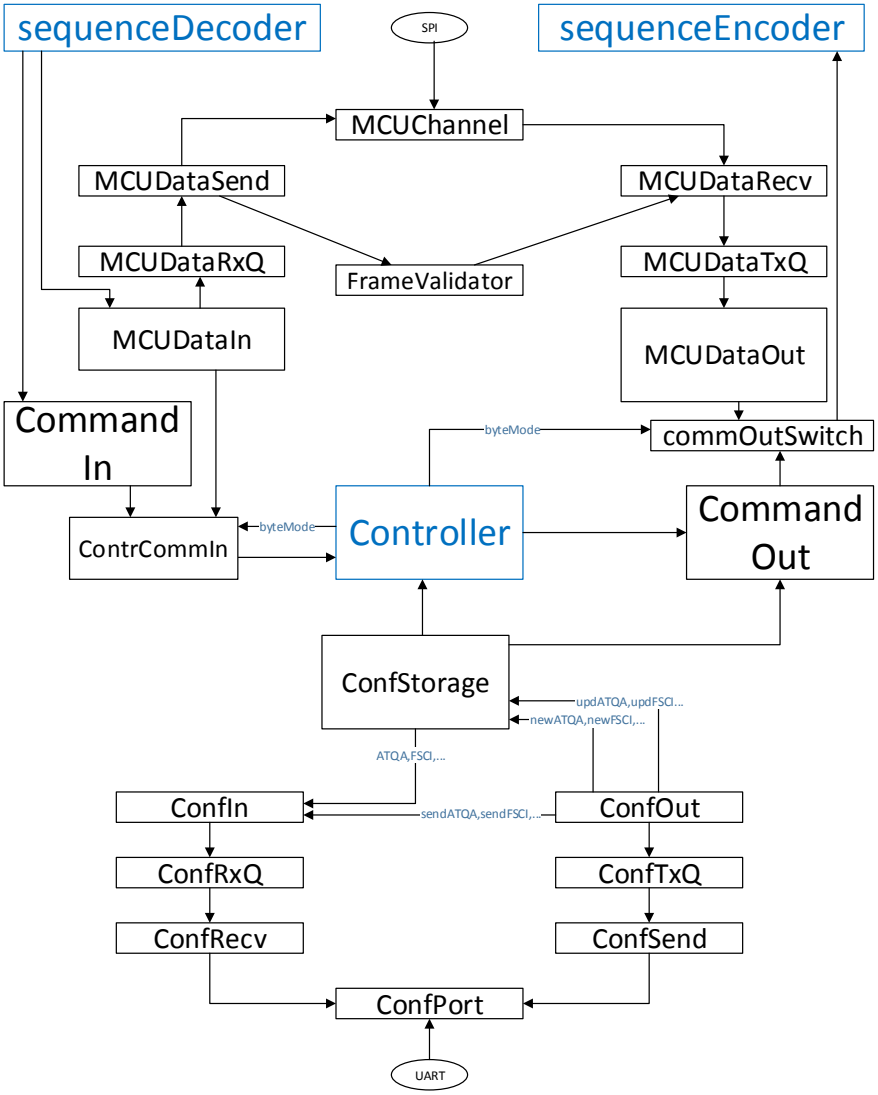


Figure 2.1: General block diagram of updated emulator

Configuration management

The emulator configuration is saved in a register file, organized in `ConfStorage` unit. To simplify emulator usage, default configuration have been hard-coded inside. It is selected after reset.

Currently, all configuration parameters (except for `FSDI`) are updated via UART. The exception was made because `FSDI` depends on a card reader and is set during ISO protocol initialization.

The `ConfOut` module handles configuration updates. This process is separated into two phases: first, `newPARAM` and `changePARAM` signals are set, where `PARAM` is name of a parameter, causing `ConfStorage` to replace saved value. After that the `sendPARAM` signal is set, telling `ConfIn` to generate reply frame with newly set value.

Configuration frames are parsed in `ConfRecv` and constructed in `ConfSend` modules. After reading frame length from the lower half-byte of the header, they wait for (or, in case of `ConfSend`, repeatedly send) respective amount of bytes and store the next frame to (or take from) respective queue.

`ConfPort` module is just an RS232 implementation. It was not written as a part of this work, but rather copied from this web page [6].

Data path modifications

The addition of an alternative data path required couple of adjustments to its surrounding units (`sequenceDecoder`, `controller` and `sequenceEncoder`). The biggest of them were made to the `controller` and are connected to data path selection.

The `byteMode` output was added. After reset, is is set to `'0'` and the emulator uses old data path, till the `controller` arrives to either `ACTIVE` or `ACTIVEStar` state. At this point the emulator can receive `RATS` block, which signifies start of high-level data exchange mode. After initialization procedure, `controller` enters the `SELECTED` state, where `byteMode` value is changed to `'1'`, activating high-level operations processing data path.

The `byteMode` signal not only manages `ContrCommIn` and `commOutSwith` multiplexors, altering data inputs. It also activates input parsing in `MCUDataIn` and alters behaviour of `sequenceDecoder` (instead of attempting to recieve and verify the whole ISO block, it just forwards decoded bytes to `MCUDataIn`).

MCU protocol implementation

`MCUDataIn` parses incoming ISO protocol blocks and turns them into one or more `MCUProto` frames. In case of "short" blocks, the procedure is quite straightforward – create `MCUProto` header, copy data and verify received block CRC. If the block is correct, post created frame to tx queue.

However, handling of I-blocks brings some complications. First, we cannot store the whole block on FPGA, as, according to the standard [4], it can be up to 256 bytes long. Second, we cannot know in advance when the frame would end.

As was already mentioned, the first issue is solved by implementation of so-called "cut-through" forwarding: as soon as byte buffer is filled, module goes from `S_ACCUM` to `S_CONT` state and posts frame with this buffer contents to the queue, returning for the next data chunk to `S_ACCUM` afterwards.

The second factor – inability to predict end of I-block in advance – becomes an issue because it can lead to accidental inclusion of CRC bytes into computed sum, as it is calculated in parallel to data receiving. The solution to this is introduction of LSR module – shift register, which allows us to delay input data processing by 2 bytes (CRC length), synchronizing `commEnd` signal with end of application data, just before CRC.

The `MCUDataOut` module transforms received `MCUProto` frames back into ISO protocol blocks. After taking a new frame from queue it generates ISO block header first, basing on frame's `OpCode` (and, in case of I and S-blocks, frame's `BNO`) and sends it to `sequenceEncoder` module. After that it proceeds with frame data, simultaneously calculating CRC of a new block, which is sent immediately after data end.

As `sequenceEncoder` doesn't provide any indication on its readiness to send another byte, we had to create new process in main module, which analyses `sequenceEncoder` output and sets availability signal, if this output takes "neutral" value. This flag is saved into 1-bit latch, to avoid excessive critical paths lengthening and make device significantly faster. Without the latch, design have synthesized for a maximal clock rate of approximately 37 MHz, while after latch addition it started to synthesize on approximately 79 MHz.

`MCUQueue` is a simple byte-oriented queue, used to store both configuration and `MCUProto` frames. It is implemented over ring buffer. Both queue length and size of one element can be changed with generic parameter.

The `MCURecv` module assembles `MCUProto` frames from bytes received on `MCUChannel` rx interface, validates received frames and if necessary requests their confirmation.

The module receives first 2 header bytes, then extracts length of frame data from respective header field and saves it to a buffer. After that, it expects 2-byte CRC, which it compares with actual sum, calculated in parallel to frame reception. If received frame is invalid, the unit discards it and enters standby state. Otherwise the frame is saved into rx queue.

For the purpose of protocol state control, it also sets several frame indication signals. These signals are: `rxReady` (fires for 1 tick, when the frame finishes processing), `rxSumOk` (`'1'` if recieved CRC have matched calculated sum), `rxFNO` (Number of frame, we've just received) and `confRequired` (`'1'` if frame require immediate confirmation – i.e. if it is `ISO_I_CONT` or `ISO_ICHAIN_CONT`).

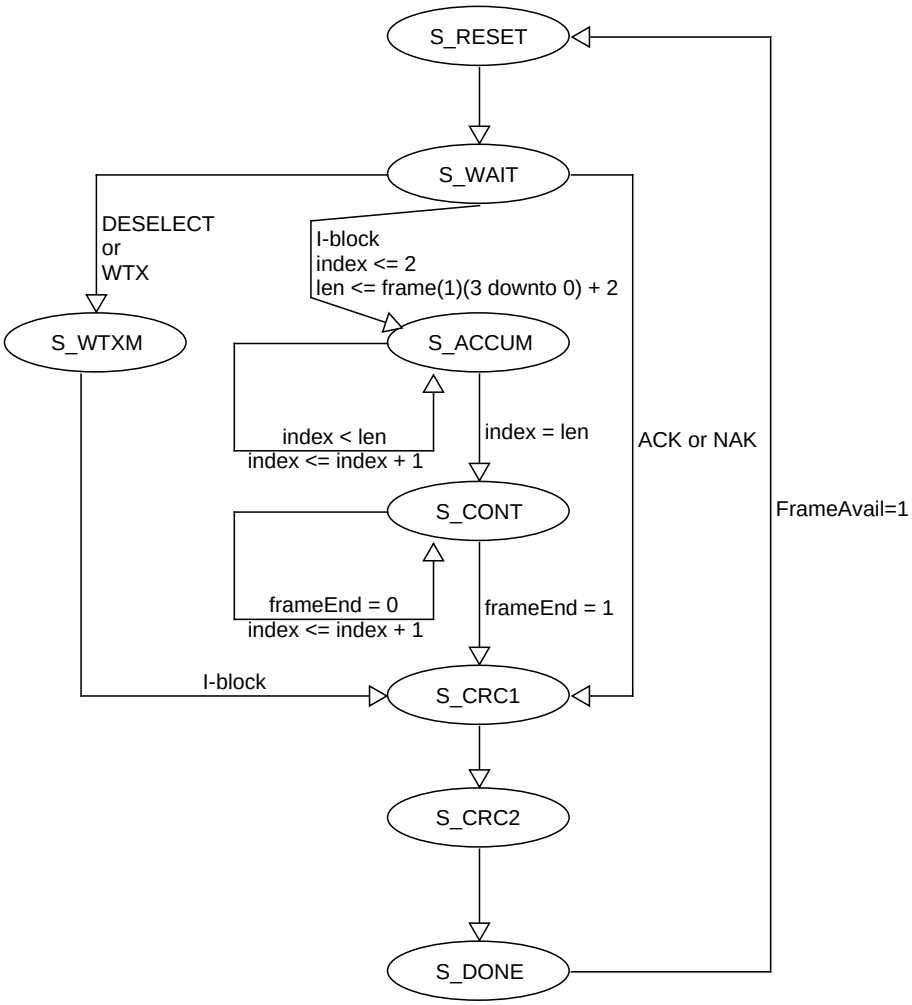


Figure 2.2: MCUDataIn state diagram

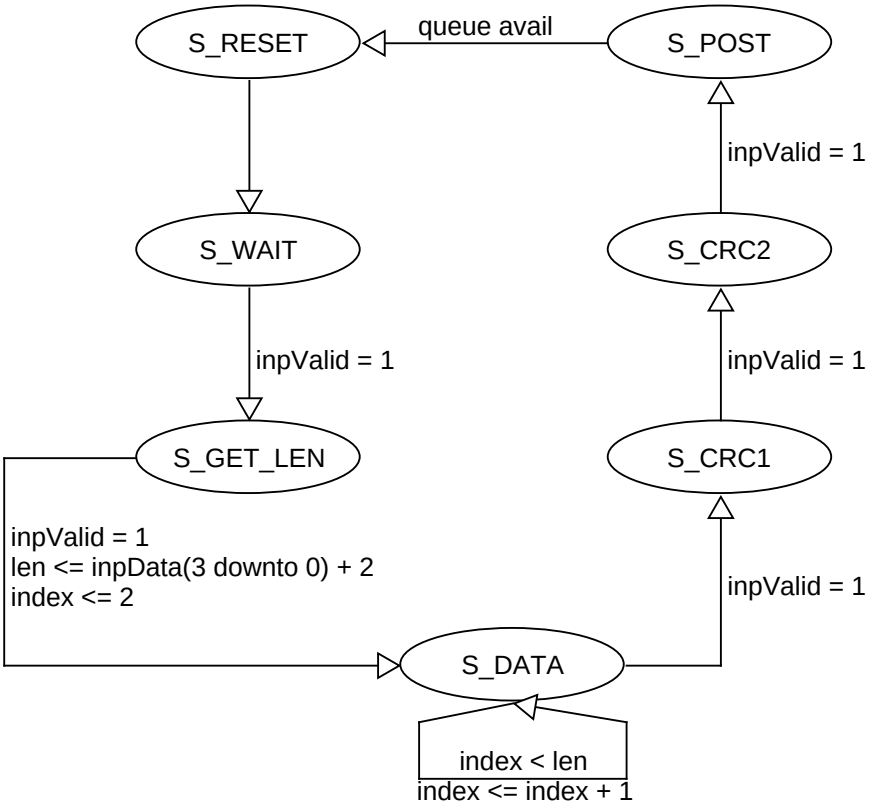


Figure 2.3: MCURecv state diagram

`FrameValidator` module handles transmission errors. Not only it checks for validity of received frame's `FNO`, but it also manages frame confirmations, scheduling their sending by `MCUSend`. Moreover, it finds out whether or not the last sent frame was confirmed and gives this information to `MCUSend`.

The `MCUSend` sends `MCUProto` frames to `MCUChannel` tx interface.

Frames to send are either taken from tx queue, or generated at place. The latter happens when either `sendMCUConf` or `sendMCUNConf` input signals fire. In this case `MCU_CONF` frame with either correct `FNO` (in case of `sendMCUConf`), or frame with inverted `FNO` (for `sendMCUNConf`) is generated. Afterwards transmission process is identical to the one, applied to frames, taken from tx queue.

The unit sends 2-byte header first, then `DataLen` bytes of frame data and finishes transmission by 2-byte CRC, which is calculated during transmission of the previous bytes.

When frame is sent, the module checks its `OpCode` and, if the frame requires immediate confirmation as `ISO_I_CONT` or `ISO_ICHAIN_CONT`, the module enters

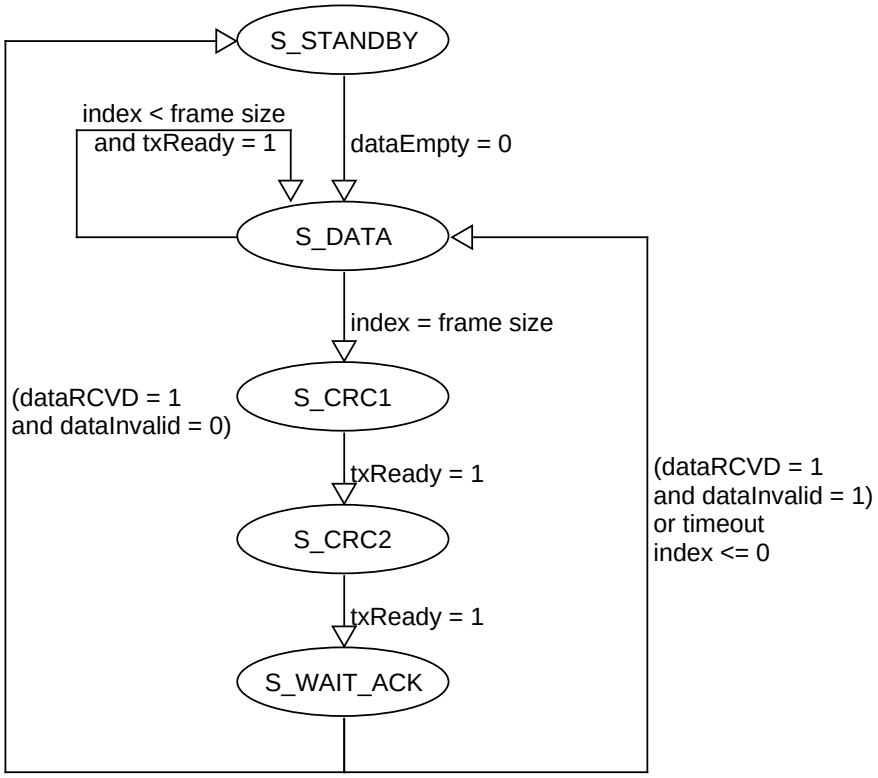


Figure 2.4: MCUSend state diagram

`S_WAIT_CONF` state. In this state timeout counter is started, ticking with each SPI `SCLK` pulse. The timeout length can be set by changing `MCUTimeout` configuration parameter. If a correct confirmation (indicated by `frameConfirmed` signal) is received before the timeout is up, the module enters standby state. Otherwise, the last frame is resent.

`MCUChannel` is an SPI master implementation. It was not created as a part of this work, but rather copied from the web. You can find original page at [7].

C library for MCU protocol support

This library – `libmcusupport` – provides high level interface, which bears some similarity to POSIX sockets networking API. For the sake of simplicity, library provides only blocking calls. It was written to be usable even in bare-metal firmware, so it depends only on a limited subset of C standard library,

Function	Description
void mcu_init(mcu_ctx_t*)	Initialize context
void mcu_connect(mcu_ctx_t*)	Wait for card selection
int mcu_isr_rx(mcu_ctx_t*, uint8_t*)	Receive next byte from SPI ISR
int mcu_isr_tx(mcu_ctx_t*, uint8_t*)	Send next byte to SPI ISR
int mcu_recv(mcu_ctx_t*, uint8_t*, size_t)	Receive data to buffer
int mcu_send(mcu_ctx_t*, uint8_t*, size_t)	Send data from buffer
int mcu_wtx(mcu_ctx_t*, uint8_t)	Ask for wait time extension

Table 2.1: List of libmcusupport functions

and doesn't rely on any system services (such as dynamic memory allocation). Small library size and very wide spectre of target environments make distribution in source code the most viable solution. As it consists of only one header and one source file, it can be included directly into a project. The library requires C99 compiler, as it actively uses fixed-size integer types from `stdint.h`. The library uses reference implementation of CRC_A algorithm, taken from standard [2] with minor modifications (data types replacement and removal of CRC_B generation).

As the library doesn't use dynamic memory allocation, all data buffers have fixed size. While small buffers would severely limit maximal size of data, that can be processed, excessively large buffers would waste memory, which can be a very limited resource, especially in embedded environment. To mitigate this problem, buffers' sizes were parametrized by `CFG_MAX_FRAMES` configuration constant, which is located in a library header and can be edited by user.

The library has 2-level architecture: `mcu_recv`, `mcu_wtx` and `mcu_send` functions make so-called *"user"* level, permitting him to exchange data with FPGA and, as result, PCD. All protocol operation rules and error checking procedures are implemented here. The other, *"transfer"*, level consists of 2 functions: `mcu_isr_rx` and `mcu_isr_tx`. These functions manage actual data reception and transmission of data bytes. As their names imply, they should be called from, respectively, receive and transmit ISRs of SPI interface.

The `mcu_ctx_t` is a connection state variable. All current state variables, as well as large data buffers (for example, currently received data) are stored here. There are two reasons for this: reentrability and stack size limitations. Storage of all state variables, which can be used between several function calls, in a structure allows us to avoid usage of global variables, thus making library reentrable.

The second reason – stack size limitations – is more important for storage of data buffers. Many microcontrollers have very small stacks. For example, ATMega 328p stack size is less than 1 kB. By placing byte buffers into struc-

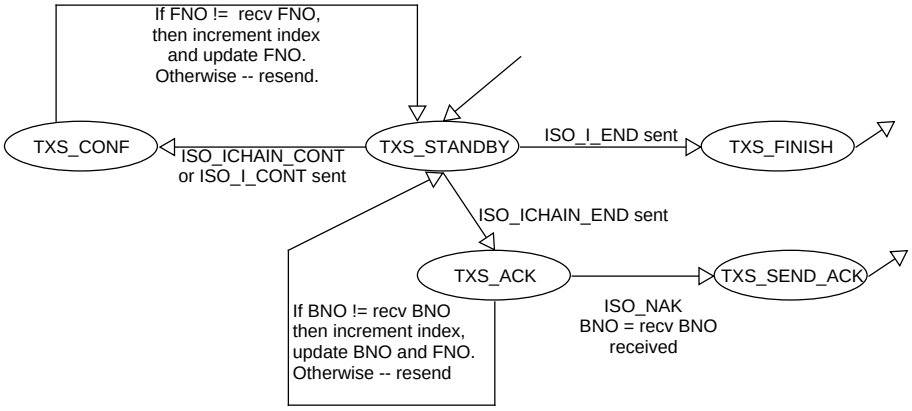


Figure 2.5: Simplified state diagram of mcu_send()

Name	Value	Description
INF_DESELECT	-1	Card was deselected by reader
ERR_OP_UNKN	-2	Unknown protocol operation
ERR_NO_CTX	-3	ctx is NULL
ERR_FAIL	-4	Internal failure

Table 2.2: List of libmcusupport error codes

ture, allocated by user, we allow him to choose data location. For example, structure can be declared as a static variable, or allocated on heap.

The connection is established by `mcu_connect()` function. It properly initializes passed `mcu_ctx_t` structure and waits for `ISO_SELECT` frame to arrive. After reading `FSDI` from incoming frame, this function sends reply (`MCU_CONF`) and exits.

From this moment on, the user can call `mcu_recv()` to get ISO I-block contents. This function would wait for incoming data, assemble data from received sequence of `MCUProto` frames and return length of received data. In case of errors, one of error codes is returned. Error codes are encoded as negative integers.

If incoming I-blocks are chained, `mcu_recv()` will handle the whole chain and assemble data, received from all its blocks into one continuous buffer.

To send reply, user should call `mcu_send()` function. It would separate data into a set of `MCUProto` frames and keep track of transaction correctness. If user will provide more data, than one I-block can carry, it would be sent as stream of `MCUProto` frames, that will generate sequence of chained I-blocks.

To simplify this function design, so-called ”data map” is constructed before

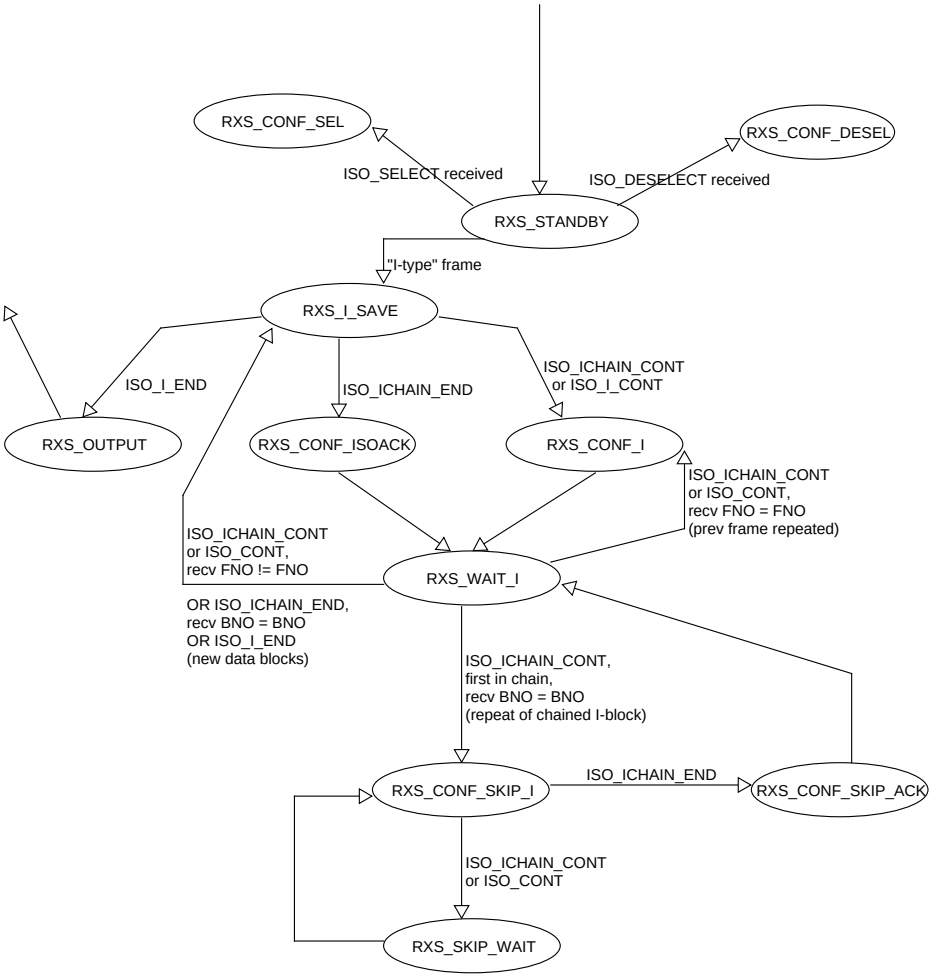


Figure 2.6: Simplified state diagram of mcu_recv()

sending data. The data map sets separation of data block into `MCUProto` frames. It is an array of `structs`, where each element corresponds to one frame, containing its type (`CONT`/`END` and `CHAINED`/usual) and length of chunk to be sent with it. Basing on this information, `mcu_send()` can easily determine borders of data chunk, currently transmitted.

Both ISO and `MCUProto` protocol operation rules are implemented in these two functions, enabling them to deal with necessary confirmations and error handling in a transparent way.

Instead of replying with data, the user can ask `PCD` for time extension, by calling `mcu_wtx()`. The function second parameter is ISO protocol `wtxm` byte, coding of which is described in the fourth part of standard [4]. This function

sends `ISO_WTX` frame and waits till `ISO_WTX` reply. If any other frame arrives, the function returns with error code of `-1`.

Finaly, `mcu_isr_rx()` and `mcu_isr_tx` functions are meant to be called inside of ISR. These functions store or, respectively, take data byte from the frame currently received or sent. By this they serve as a bridge between platform-specific SPI handling and generalised library.

Chapter 3

Testing

It would be nice to have full integration tests, but due to the lack of time the testing procedure have been shortened. Running full integration test would require quite a complex setup, including usage of programmable card reader. ”In-circuit” input generation, where ”fake” input signals would be generated on board button press, haven’t been implemented as well: high degree of integration haven’t allowed us to implement that system in time. As result, all testing efforts were done on a level of separate units.

Behavioural simulation of VHDL entities

All modules, introduced by this work have been tested by a behavioural modelling, using the ISim simulator of version P.20131013, which is distributed together with Xilinx ISE 14.7.

We have used so-called ”grey box” testing method, where tests implementation is dependent on knowledge of module’s internal structure. For example, minimal delays between input updates take internal state changes of the tested

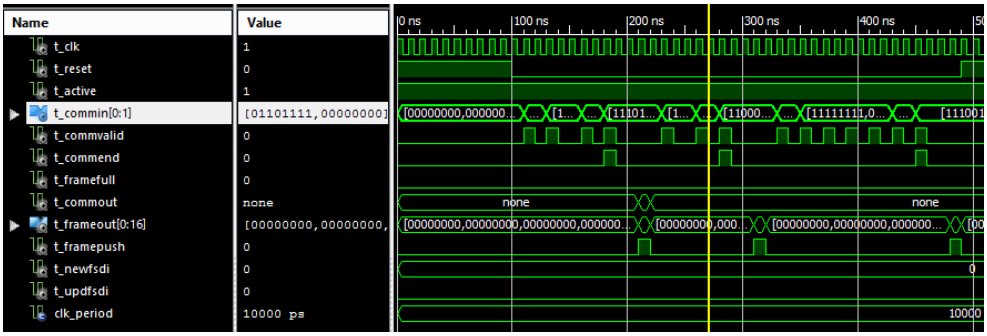


Figure 3.1: Simulation of MCUDataIn module

unit into an account. Please note, that in reality input change frequencies are less frequent than design clock by two orders of magnitude (for example, SPI clock frequency is 250 kHz, while system clock works at 50 MHz). Accurate modelling of this difference would only increase simulation time, without any practical gains.

Due to input variety, exhaustive testing seems to be impractical. As result, several test vectors were chosen for each unit. Representation of major paths in each module was the main selection criteria.

Testing the protocol support library

Due to the relatively small size of the library, it can be sufficiently tested, using only unit tests. Usage of the protocol state structure have allowed us to easily replace either `mcu_isr_tx()/mcu_isr_rx()` level, or `mcu_send()/mcu_recv` level, respectively.

However, protocol state structure contains only last received (or sent) data frame. When *"user"* level function, such as `mcu_recv()`, consumes current data block, it relies on controller interrupts to pause this function execution and to update received data. To emulate this situation on desktop pc, where the testing is conducted, some tests were implemented in two threads, where one thread was executing the tested function, while another was updating this function inputs. We will present test suite report here:

```
Initializing test environment
Component testing:
Test 1 (Connection):  Ok
Test 2 (recv interrupt):  Ok
Test 3 (send interrupt):  Ok
Test 4 (Receive 1 frame I-block:  Ok
Test 5 (Send 1 frame I-block:  Ok
Test 6 (Receive multiframe I-block:  Ok
Test 7 (Send multiframe I-block:  Ok
Test 8 (Deselect):  Ok
```

Chapter 4

Future work

The system can use more deep and complex testing. Only typical scenarios have been checked at this moment. Examination of corner cases and load testing can improve system stability and robustness by a large margin. Not only the implementation should be verified, but the `MCUProto` itself can be validated to be able to handle all possible problems scenarios.

Another area of improvement lies in ISO protocol feature support. The optional PPS request/reply part of protocol initialization is not supported at the moment. According to the standard, it is used to set several transmission-related options, such as default FWT or CID header support. Support of operation in environment, where multiple cards are selected at the same moment should also increase usability of the emulator.

The microcontroller part can also be worked on. Attempt to implement an actual proprietary high-level protocol can verify suitability of both `mcusupport` library and `MCUProto` forwarding protocol for this purpose, leading to possible optimizations or interface changes.

Chapter 5

Conclusion

Support of forwarding high-level ISO protocol blocks was added to FPGA. For this purpose, a special protocol (`MCUProto`) was designed. Even though some of ISO/IEC 14443-4 features are not supported (for example, PPS request/reply procedure), core of the standard – support of half-duplex data exchange protocol – was implemented.

For the microcontroller side a C protocol support library (`libmcusupport`) was implemented. Depending only on a small subset of C standard library, `libmcusupport` can be used on both bare-metal and RTOS-based firmwares.

SPI was chosen as a physical interface between FPGA and MCU boards for its speed and simplicity.

It would be good to test both FPGA and library more rigorously, but due to the lack of time we had to skip some tests, as they required complex environment or heavy code modification.

The Emulator configuration have been made available on RS232 interface, where user can query or change separate parameters, using a simple protocol.

Even though there is a lot to improve: reliability of implementation, data forwarding protocol robustness and degree of standard support; this seems to be a good starting point for further experiments in different proprietary smart card protocols emulation.

Bibliography

[1] Jeřábek, I. S. *Emulátor bezkontaktní čipové karty v FPGA*. Master's thesis, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

[2] International Organization for Standardization. *ISO/IEC 14443 – Identification cards – Contactless integrated circuit cards – Proximity cards – Part 3: Initialization and anticollision*. First edition, 2001.

[3] AN10833 MIFARE Type Identification Procedure. `http://www.nxp.com/documents/application_note/AN10833.pdf`, [rev. 11 July, 2016], [cit. 5 Mar 2017].

[4] International Organization for Standardization. *ISO/IEC 14443 – Identification cards – Contactless integrated circuit cards – Proximity cards – Part 4: Transmission protocol*. First edition, 2001.

[5] International Organization for Standardization. *ISO/IEC 7816 – Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*. Third edition, 2006.

[6] RS232 module. `https://edux.fit.cvut.cz/courses/MI-BHW/_media/tutorials/01/rs232.zip`, [cit. 5 Mar 2017].

[7] Serial Peripheral Interface (SPI) Master (VHDL). `https://eewiki.net/pages/viewpage.action?pageId=4096096`, [ver. 1.1], [cit. 5 Mar 2017].

Appendix A

Acronyms

ACK ACKnowledge

ATQA Answer To reQuest, Type A

ATS Answer To Select

BNO Block Number

CID Card IDentifier

ConfKey Configuration Key

ConfProto Emulator Configuration Protocol

CRC Cyclic Redundancy Check

DataLen Data Length

etu Elementary Transmission Unit

fc Carrier frequency

FNO Frame Number

FPGA Field-Programmable Gates Array

FWT Frame Waiting Time

FSC Frame Size for proximity Card

FSCI Frame Size for proximity Card Integer

FSD Frame Size for proximity coupling Device

FSDI Frame Size for proximity coupling Device Integer

ISO International Standardization Organisation

ISR Interrupt Service Routine

MCU Microcontroller Computing Unit

MCUProto ISO data forwarding protocol

NAD Node ADdress

NAK Negative AcKnowledge

OpCode Operation Code

OSI Open Systems Interconnection basic reference model

PCD Proximity Card Device

PICC Proximity Identification Card

PPS Protocol and Parameter Selection

RAM Random Access Memory

RATS Request to Answer To Select

RFU Reserved for Future Use

RS232 Recommended Standard 232

SAK Select AcKnowledge

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver-Transmitter

ValueLen Value Length

WTX Wait Time eXtension

WTXM Wait Time eXtension Multiplier

Appendix B

Contents of enclosed CD

- `readme.txt` ..... CD contents description
- `sources` ..... Thesis implementation directory
  - `impl` ..... Implementation of emulator for FPGA
  - `mcusupport` ..... C library for MCUProto support
  - `tex` ..... Directory with LaTeX source codes of the thesis
- `thesis.pdf` ..... Thesis text in PDF format